Functions are “first-class” values
• Arguments, return values, bindings ...
• What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Useful if parameterized functions can be passed to, hence used/called by other functions...

News
• PA4 is up
  - Due 5/6
  - Week *AFTER* midterm

• Midterm 4/28 (?)
  - In class, open book etc.
  - Practice materials on Webpage
  - Depends on PA4 ...

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions
Why take functions as input?

let rec evens l =
  match l with
  | [] -> []
  | h::t -> if is_even h then h::(evens t) else evens t

let rec lessers x l =
Why take functions as input?

let rec evens l =
  match l with
  [] -> []
  | h::t -> if is_even h then h::(evens t) else evens t

let rec lessers x l =
  match l with
  [] -> []
  | h::t -> if h<x then h::(lessers x t) else lessers x t

Factoring and Reuse

let rec lessers x l =
  match l with
  [] -> []
  | h::t -> if h<x then h::(lessers x t) else lessers x t

let rec filter f l =
  match l with
  [] -> []
  | h::t -> if (f h) then h::(filter f t) else filter f t

“Factor” code:

- Generic pattern
- Specific instance

let lessers x l =
  filter (fun i -> i<x) l

let lessers x l =
  filter (fun i -> i<x) l
Factoring and Reuse

```
let rec filter f l = 
  match l with 
    [] -> [] 
  | h::t -> if (f h) then h::(filter f t) else filter f t
```

“Factor” code:
- Generic pattern
- Specific instance

```
let evens l = 
  filter is_even l
```

Encoding Patterns as functions

```
let rec filter f l = 
  match l with 
    [] -> [] 
  | h::t -> if (f h) then h::(filter f t) else (filter f t)
```

```
let neg f = fun x -> not (f x)
let partition f l= (filter f l, filter(neg f) l))
```
Encoding Patterns as functions

```
let rec filter f l =
  match l with
    []   -> []
  | h::t -> if (f h) then h::(filter f t)
      else (filter f t);
```

```
let neg f = fun x -> not (f x)
let partition f l= (filter f l, filter(neg f) l))
```

**filter, neg, partition**: higher-order functions

- Take a any tester as argument!

Iteration Pattern

```
let rec listUppercase xs =
  match xs with
    []   -> []
  | h::t -> (uppercase h)::(listUppercase t)
```

```
let rec listSquare xs =
  match xs with
    []   -> []
  | h::t -> (h * h)::(listSquare t)
```

```
let addPair (x,y) = x + y
let rec listAddPair xs =
  match xs with
    []     -> []
  | (hx,hy)::t ->(addPair (hx,hy))::(listAddPair t)
```

Iteration Pattern

```
let rec listUppercase xs =
  match xs with
    []   -> []
  | h::t -> (uppercase h)::(listUppercase t)
```

```
let rec listSquare xs =
  match xs with
    []   -> []
  | h::t -> (h * h)::(listSquare t)
```

```
let addPair (x,y) = x + y
let rec listAddPair xs =
  match xs with
    []     -> []
  | (hx,hy)::t ->(addPair (hx,hy))::(listAddPair t)
```
let rec listUppercase xs = 
  match xs with 
  | [] -> [] 
  | h::t -> (uppercase h)::(listUppercase t)

let rec map f l = 
  match l with 
  | [] -> [] 
  | (h::t) -> (f h)::(map f t)

let listUpperCase l = map upperCase l
### Higher-order functions: map

**Type says it all!**
- Applies “f” to each element in input list
- Makes a list of the results

```ocaml
let rec map f l = 
  match l with 
    [] -> [] 
 |  (h::t) -> (f h)::(map f t)
```

**Examples**
- `let listUpperCase l = map upperCase l`
- `let listSquare l = map (fun x -> x*x) l`
- `let listAddpair l = map (fun (x,y) -> x+y) l`

---

### Factoring Iteration w/ “map”

```ocaml
let rec map f l = 
  match l with 
    [] -> [] 
 |  (h::t) -> (f h)::(map f t)
```

**Examples**
- `let listUpperCase l = map upperCase l`
- `let listSquare l = map (fun x -> x*x) l`
- `let listAddpair l = map (fun (x,y) -> x+y) l`
Factoring Iteration w/ “map”

“Factored” code:
• Reuse iteration template
• Avoid bugs due to repetition
• Fix bug in one place!

Another pattern: Accumulation

let max x y = if x > y then x else y ;
let listMax l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (max cur h) t
  in
    helper 0 l;;

let concat l =
  let rec help cur l =
    match l with
    [] -> cur
    | h::t -> help (cur^h) t
  in
    helper "" l;;

let rec map f l =
  match l with
    [] -> []
  | (h::t) -> (f h)::(map f t)
Let rec fold f cur l =
  case l of
    [] -> cur
    | h::t -> fold f (f cur h) t

What is: fold f base [v1;v2;…;vn] ?

  f(...( f( ,v3),vn)
  f(base,v1)
  f( ,v2)
  f( ,v3)
  f(...( ,vn)
What's the pattern? Tail Rec?

Examples of fold

```plaintext
let listMax = fold max 0
```

```plaintext
let concat = fold (^) ""
```

```plaintext
let multiplier =
```

Pick correct base case!

What does this do?

```plaintext
let f l = fold (::) [] l
```
Examples of fold

```
let f l = fold (::) [] l
```

Funcs taking/returning funcs

Identify common computation “patterns”
• Filter values in a set, list, tree ...
• Iterate a function over a set, list, tree ...
• Accumulate some value over a collection

Pull out (factor) “common” code:
• Computation Patterns
• Re-use in many different situations

Another fun function: “pipe”

```
let pipe x f = f x
```

Another fun function: “pipe”
Another fun function: “pipe”

```plaintext
let pipe x f = f x

let (|>) x f = f x
```

Compute the sum of squares of numbers in a list?

```plaintext
let sumOfSquares xs =
   xs |> map (fun x -> x * x)
      |> fold_left (+) 0
```

Tail Rec?
**Funcs taking/returning funcs**

Identify common computation “patterns”
- Filter values in a set, list, tree ...
- Convert a function over a set, list, tree ...
- Iterate a function over a set, list, tree ...
- Accumulate some value over a collection

Pull out (factor) “common” code:
- Computation Patterns
- Re-use in many different situations

**Functions are “first-class” values**

- Arguments, return values, bindings ...
- What are the benefits?

Parameterized, similar functions (e.g. Testers)

Creating, (Returning) Functions

Using, (Taking) Functions

Compose Functions:
Flexible way to build Complex functions from primitives.

**Data Structure Library**

Provides meta-functions: map, fold, filter to traverse, accumulate over lists, trees etc.
Meta-functions don’t need client info (tester ? accumulator ?)

**Data Structure Client**

Uses list
Uses meta-functions: map, fold, filter
With locally-dependent funs (lt h), square etc.
Without requiring implement. details of data structure

**Functions are “first-class” values**

- Arguments, return values, bindings ...
- What are the benefits?

- Each part only needs local information

**Funcs taking/returning funcs**
Functions are “first-class” values

- Arguments, return values, bindings ...
- What are the benefits?

Everywhere:
- Javascript,
- Google/Mapreduce, Yahoo/Hadoop,
- C++0x

What is the deal with ’a?

These meta-functions have strange types:

map: \( (\text{'a} \to \text{'b}) \to \text{'a list} \to \text{'b list} \)

filter: \( (\text{'a} \to \text{bool}) \to \text{'a list} \to \text{'a list} \)

Why?

Polymorphism

- Poly = many, morph = kind

\[
\text{let swap (x,y) = (y,x)} \quad \text{'a * 'b -> 'b * 'a}
\]

- ’a and ’b are type variables!
- For-all types: \( \text{For all 'a, 'b: 'a * 'b -> 'b * 'a} \)

- ’a,’b can be instantiated with any type:
  - w/ \text{int,string : int * string -> string * int}
  - w/ \text{char, int list : char * int list -> int list * char}
  - w/ \text{int->int , bool : (int -> Int) * bool -> bool *(int ->int)}
Instantiation at Use

\[
\begin{align*}
\text{map:} & \quad \text{('a \, 'b) \, 'a list \, 'b list} \\
\end{align*}
\]

\[
\begin{align*}
\text{f : int -> int} & \quad \text{first arg. of map: a->b} \\
\text{Instantiated: 'a with int, 'b with int} \\
\end{align*}
\]

\[
\begin{align*}
\text{f : string -> string} & \quad \text{first arg. of map: a->b} \\
\text{Instantiated: 'a with str, 'b with str} \\
\end{align*}
\]

\[
\begin{align*}
\text{let } f \, x &= \, x + 10; \\
\text{let } fm &= \text{map} \, f; \\
\end{align*}
\]

\[
\begin{align*}
\text{let } f \, x &= \, x^\text{“ like”}; \\
\text{let } fm &= \text{map} \, f \, \text{[“cat”; “dog”; “burrito”]}; \\
\end{align*}
\]
Instantiation at Use: be careful

map: \((a \to b) \to [a\;\text{list}] \to [b\;\text{list}]\)

let \(f\) \(x\) = \(x^\text{"like"};\);
let \(fm\) = \(\text{map } f\) \([1;2;3;4]\);

\(\text{String list } \to \text{string list}\)

\(f\) : \(\text{string } \to \text{string}\) \quad \text{first arg. of map: } a\to b

\text{Instantiated: } a\text{ with str, } b\text{ with str}

\text{So, list must be } a\text{ list } = \text{string list!}

Polymorphic ML types

- Poly = \textit{many}, morph = \textit{kind}

- Possible ML types:
Polymorphic ML types

• Poly = *many*, morph = *kind*

• Possible ML types:

  \[ \text{tv} = 'a \mid 'b \mid 'c \mid \ldots \]

  \[ T = \text{int} \mid \text{bool} \mid \text{string} \mid \text{char} \mid \ldots \]

  \[ T_1 \times T_2 \times \ldots T_n \mid T_1 \rightarrow T_2 \mid \text{tv} \]

• Implicit for-all at the “left” of all types
  - Never printed out

map: \( ('a \rightarrow 'b) \rightarrow 'a \text{ list} \rightarrow 'b \text{ list} \)
Polymorphism enables Reuse

- Can reuse generic functions:
  
  - `map` : `'a * 'b -> 'b * 'a`
  - `filter` : `(a -> bool) -> 'a list -> 'a list`
  - `rev` : `'a list -> 'a list`
  - `length` : `'a list -> int`
  - `swap` : `'a * 'b -> 'b * 'a`
  - `sort` : `(a -> a -> bool) -> 'a list -> 'a list`
  - `fold` : ...

- If function (algorithm) is independent of type, can reuse code for all types!

Not just functions …

- Data types are also polymorphic!

  - `type 'a list = 
    Nil
    | Cons of ('a * 'a list)`

Not just functions …

- Data types are also polymorphic!

  - `type 'a list = 
    Nil
    | Cons of ('a * 'a list)`
• Data types are also polymorphic!

\[
\text{type } \tau \text{ list } = \\
\quad \text{Nil} \\
\mid \text{Cons of } (\tau \times \tau \text{ list})
\]

• Type is instantiated for each use:

\[
\begin{align*}
\text{Cons}(1, \text{Cons}(2, \text{Nil})) & : \text{int list} \\
\text{Cons}(\text{“a”}, \text{Cons}(\text{“b”}, \text{Nil})) & : \text{string list} \\
\text{Cons}((1,2), \text{Cons}((3,4), \text{Nil})) & : (\text{int} \times \text{int}) \text{ list} \\
\text{Nil} & : \tau \text{ list}
\end{align*}
\]
Datatypes with many type variables

- Multiple type variables

define ('a,'b) tree =
    Leaf of ('a * 'b)
  | Node of ('a,'b) tree * ('a,'b) tree
Datatypes with many type variables

- Multiple type variables

```ocaml
type ('a,'b) tree =
  Leaf of ('a * 'b)
| Node of ('a,'b) tree * ('a,'b) tree
```

- Type is instantiated for each use:

```ocaml
Leaf("joe",1) :
Leaf("william",2) :
Node(...,...) :
Node(Leaf("joe",1),Leaf(3.14, "pi")):
```

Polymorphic Data Structures

<table>
<thead>
<tr>
<th>key, data</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a list</td>
</tr>
<tr>
<td>('a , 'b) tree</td>
</tr>
<tr>
<td>('a , 'b) hashtbl</td>
</tr>
</tbody>
</table>

Datatypes with many type variables

- Multiple type variables

```ocaml
type ('a,'b) tree =
  Leaf of ('a * 'b)
| Node of ('a,'b) tree * ('a,'b) tree
```

- Type is instantiated for each use:

```ocaml
Leaf("joe",1) : (string,int) tree
Leaf("william",2) : (string,int) tree
Node(...,...) : (string,int) tree
Node(Leaf("joe",1),Leaf(3.14, "pi"))
```

Polymorphic Data Structures

<table>
<thead>
<tr>
<th>'a list</th>
</tr>
</thead>
<tbody>
<tr>
<td>('a , 'b) tree</td>
</tr>
<tr>
<td>('a , 'b) hashtbl</td>
</tr>
</tbody>
</table>
Polymorphic Data Structures

- Container data structures independent of type!
- Appropriate type is instantiated at each use:
  
  - 'a list
  - ('a, 'b) tree
  - ('a, 'b) hashtbl ...

- Appropriate type instantiated at use
  - No unsafe casting as in C/C++/Java
Polymorphic Data Structures

- Container data structures independent of type!
- Appropriate type is instantiated at each use:
  - 'a list
  - ('a, 'b) tree
  - ('a, 'b) hashtbl ...

- Appropriate type instantiated at use
  - No unsafe casting as in C/C++/Java

- Static type checking catches errors early
  - Cannot add int key to string hashtable

Polymorphic Data Structures

- Container data structures independent of type!
- Appropriate type is instantiated at each use:
  - 'a list
  - ('a, 'b) tree
  - ('a, 'b) hashtbl ...

- Appropriate type instantiated at use
  - No unsafe casting as in C/C++/Java

- Static type checking catches errors early
  - Cannot add int key to string hashtable

Generics: in Java, C#, VB (borrowed from ML)

Other kinds of polymorphisms

- That was OCaml...
- But what about other kinds of polymorphisms..
Other kinds of polymorphisms

• Sub-type polymorphism
  void f(Shape s)
  - Can pass in any sub-type of Shape
Other kinds of polymorphisms

- Sub-type polymorphism
  
  ```
  void f(Shape s)
  - Can pass in any sub-type of Shape
  ```

- Parametric polymorphism
  
  ```
  void proc elems(list[T])
  - can pass in ANY T
  ```

Other kinds of polymorphisms

- Sub-type polymorphism
  
  ```
  void f(Shape s)
  - Can pass in any sub-type of Shape
  ```

- Parametric polymorphism
  
  ```
  void proc elems(list[T])
  - can pass in ANY T
  ```
Other kinds of polymorphisms

- Sub-type polymorphism
  ```java
  void f(Shape s)
  - Can pass in any sub-type of Shape
  ```

- Parametric polymorphism
  ```java
  void proc_elems(list[T])
  - can pass in ANY T
  - this is the kind in OCaml!
  ```

- Bounded polymorphism
  ```java
  void proc_elems(list[T]) T extends Printable
  - Hey... isn’t this subtype polymorphism?
  ```

- Bounded polymorphism
  ```java
  bool ShapeEq(T a, T b) T extends Shape
  - Can call on
    • (Rect, Rect)
    • (Circle, Circle)
  - But not (Rect, Circle)
Summary of polymorphism

- Subtype

- Parametric

- Bounded = Parametric + Subtype
  - In Java/C#

Back to OCaml

- Polymorphic types allow us to reuse code

- However, not always obvious from staring at code

- But... Types never entered w/ program!